

# Expr Language Reference

The Expr language is used in the Formula columns and in other places in Structure to produce calculated values based on Jira and Structure data. This reference is a detailed description of the Expr language, its syntax and the calculation rules.

To start learning Expr and what you can do with Structure Formulas, check out [Sample Formulas](#) or a more high-level [Expr Language description](#). This is an advanced material that may be useful for digging deeper, troubleshooting, or building advanced formulas. It may require some programming knowledge.



This reference contains railroad diagrams to illustrate some of the language syntax. They are intentionally simplified and should not be considered a full definition of the language.

- [Introduction](#)
  - [Expr Language Versions](#)
  - [Expressions and Values](#)
    - [Lazy Calculation](#)
    - [Displaying Calculation Result in a Formula Column](#)
  - [Comments](#)
  - [Identifiers](#)
  - [Keywords](#)
- [Values and Types](#)
  - [Undefined](#)
  - [Text](#)
    - [Text Snippets \("""\)](#)
  - [Number](#)
    - [Date and Date/Time](#)
    - [Duration](#)
    - [Boolean](#)
  - [Item](#)
    - [Special conversion of Item values](#)
    - [Special treatment of Item values by generators](#)
  - [Array](#)
    - [Creating an array](#)
    - [Accessing array elements](#)
    - [Special treatment of Array values by the Attribute Grouper generator](#)
  - [User Function](#)
  - [Key-Value Map](#)
  - [Errors](#)
- [Value Conversions](#)
  - [Text to Number Conversion](#)
  - [Conversion to Boolean: Falsy and Truthy Values](#)
  - [Text vs. Text/Joined](#)
  - [Passing Implicit User Function as a Parameter](#)
- [Variables](#)
  - [Variable to Attribute Mapping](#)
  - ["this" Variable](#)
- [Local Variables](#)
  - [Immutability](#)
- [Operators](#)
  - [Logical Negation \(NOT\)](#)
  - [Unary Sign \(+ -\)](#)
  - [Arithmetic operators \(\\* / + -\)](#)
  - [Concatenation \(CONCAT\)](#)
  - [Equality check \(= !=\)](#)
  - [Numeric Comparison \(< > <= >=\)](#)
  - [Logical operators \(AND OR\)](#)
- [Conditional Expression](#)
- [Property Access](#)
  - [Accessing Custom Fields via Properties](#)
  - [Accessing Property of Each Element in an Array](#)
- [Functions](#)
  - [System and User Functions](#)
  - [Chained Function Calls](#)
  - [Applying Functions to Arrays](#)
    - [Using an Array Function](#)
    - [Passing Array as an Argument of Text/Joined Type](#)
    - [Passing Array as an Argument of /Each Type](#)
    - [Passing Array to a User Function](#)
    - [All Other Cases](#)
- [User Functions](#)
  - [Functional Expression](#)
  - [Traditional Function Definition](#)
  - [Implicit Functional Expression \(\\$\)](#)
  - [Calling User Functions](#)
  - [Function Name Collisions](#)
- [Aggregate Functions](#)

- [Aggregate Function Modifiers](#)
- [Sharing Values Between Outer and Inner Formulas](#)
- [Using Formulas with Aggregate Functions in Generators and Transformations](#)
- [Embedded Queries](#)
  - [Using S-JQL in Generators and Transformations](#)
  - [S-JQL Query Performance](#)

## Introduction

In its simplest form, Expr language is built to be similar to Excel or Google Sheets formulas. Similarity to spreadsheet formulas, when possible, was a design goal, particularly when performing arithmetic operations, using functions, and referencing values from the spreadsheet (Jira fields or results of other formulas). The language becomes more complex once you need to work with arrays or items with properties.

Expr is a declarative, dynamically typed language with elements of functional programming.

A few properties of the language are:

- Expr is case-insensitive. Two identifiers different only in the upper or lower case will mean the same thing.
- Whitespace, including new lines, is not meaningful. It is only required to separate word operators and identifiers; in all other cases there can be an arbitrary number of whitespace symbols.
- Except in Text literals, the language supports only English (ASCII) letters, numbers and some punctuation symbols.
- Aggressive type conversion: when a function requires a certain type and the value passed is of a different type, Expr will make the best effort to convert the passed value to the required type.

## Expr Language Versions

Expr Version	Structure Version	Major Improvements
Expr 1.0	Structure 4.0	Original version introduced
Expr 1.5	Structure 4.2	Local variables, aggregate functions, comments
Expr 2.0	Structure 7.0	Arrays, Items and Properties, User Functions, Chained Function Calls, Embedded Queries, Conditional Expression, Text Snippets, Concatenation Operator

For the compatibility notes in Expr 2.0, see [Changes to Expr in Structure 7](#).

## Expressions and Values

A program written in Expr language is *an expression*. An expression is evaluated by Structure and produces a *value*. In a Formula column, the expression is calculated for each row by applying the expression to the issue or another item in that row, and the result is displayed in the corresponding Formula column cell.

Expr language is *composable*. If you have a valid expression, you can include it as a part of some other expression and it will be valid. Sometimes that will require wrapping it with parentheses.

## Lazy Calculation

Expr expressions are calculated in a lazy manner (excluding aggregate functions and embedded queries). This means that, for example, only one branch in an IF expression is going to be calculated.

## Displaying Calculation Result in a Formula Column

The [Formula Column](#) is the primary means of authoring and using formulas. It currently can display simple values, items and flat arrays. (See Values and Types below.) If the formula returns a more complex construct – nested arrays or key-value maps, the current version of the column will not be able to display it. However, it doesn't mean the formula is not getting calculated!

## Comments

At any place where a formula allows whitespace, you can use comments. Comments can span multiples lines or just one.

- Multi-line comments start with "`/*`" and end with "`*/`" and can span multiple lines. Multi-lined comments cannot be nested.
- Single-line comments start with "`//`" and continue through the end of the line.

## Identifiers

All named elements of Expr – variables, local variables, functions and others – must have a valid identifier as their name.

An identifier consists of letters (Latin alphabet only: a-z, A-Z), digits (0-9) or underscore (`_`) characters. The first character must be a letter or an underscore. An identifier must not be a keyword.



Versions 1.x of Expr also allowed the period character ( . ) to be a part of the identifier. This possibility has been removed in Expr 2.0.

## Keywords

The following keywords must not be used as identifiers.

**AND, CONCAT, ELSE, IF, NOT, OR, UNDEFINED, WITH**

## Values and Types

All expressions, when evaluated, produce either a value or an error. All values are immutable – if you need to change a value, you derive another value from it.

Each value belongs to a certain type. There are simple and complex types.

Simple types represent just one value and allow you to write literal values (constants). Simple types are:

- Undefined
- Text
- Number

Complex types represent a value that has some internal structure. Complex types are:

- Item
- Array
- User Function
- Key-Value Map

Errors are special values, but they usually cause the whole expression to return an error.



A library function may expect certain types as parameters and will produce a particular type as the result. Expr will also try to convert one type to another as needed.

## Undefined

Undefined is a special value, which represents "no value" or a missing value. For example, variable `Assignee` will have undefined value if the issue is unassigned.

Undefined value can be used instead of a value of any other type.

To use the undefined value in a formula explicitly, you can write **undefined**.



Functions can return **undefined** when the result of the function is not specified. For example, the function `IF(N = 0; "No apples"; N = 1; "One apple")` only has a specified value when N is equal to 0 or 1. If N is equal to anything else, it returns **undefined**.

## Text

A text value consists of 0 or more Unicode symbols. Its literal representation consists of the value enclosed in single quotes (') or double quotes ("). Example: **"Major"** represents text value *Major*. Similarly, **'Major'** represents the same text value.

If the text value itself contains quotes, you'll need to insert a backslash (\) before them. Example: **"Charlie \"Bird\" Parker"** represents the text value *Charlie "Bird" Parker*. Alternatively, you can use another kind of quotes to enclose the literal representation: **'Charlie "Bird" Parker'**.

If you need to use the backslash at the end of text value, you will need to insert another backslash before it. Example: **"C:\\Users\\John\\"** represents text value *C:\Users\John\*.

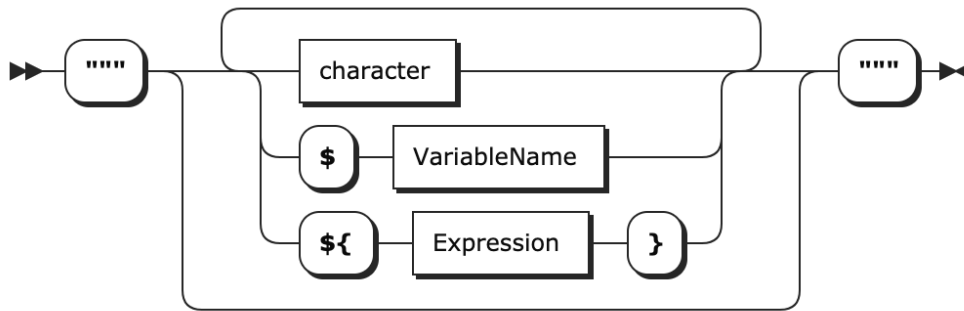
## Text Snippets ("")

Sometimes you need a long text value that contains some words and symbols, but also the value of a certain expression or a variable. One way to create it is to use the **CONCAT** function and join all the pieces of text together. A more convenient way could be writing a text snippet.

A text snippet starts with triple double quotes (""") and continues until the next triple double quotes. It includes ends of line, so you can create multiline text values with a text snippet.

Text snippets may contain special sequences that get replaced with a calculated value:

- A dollar sign followed by a variable name is replaced by that variable's value. For example, `""Assigned to: $assignee""`.
- A dollar sign followed by an Expr expression in curly braces is replaced by the value of that expression. For example, `""Total score: ${score + subtasks.score.SUM()}""`.



## Number

A number represents a single numerical value. Aside from representing some quantity, a number value can also represent a point in time or a duration of time. In this case, you can use Format settings in the [Formula column](#) to properly display the results as dates or durations.

There are two forms of literal representations of numbers:

- a whole number: `42`
- a fractional number: `0.239`



Note that only dot (.) can be used as a decimal separator. Comma (,) is used to delimit function arguments. Thus, `MAX(X, 0, 618)` will be understood as the maximum of three values: X, 0, and 618.

Group separators are not supported, so `100 000` is not a literal representation of number 100000. However, you can write a string value and explicitly call the conversion function – `NUMBER("100 000")`. See [number conversion](#).



**Technical note:** internally, numbers are represented as decimal floating-point numbers with 16 digits of precision and half-even rounding. Most of the operations are carried out in this form; however, some of the more sophisticated functions, such as `SQRT`, might first convert the numbers into binary floating-point, calculate the result and then convert it back into decimal floating-point.

## Date and Date/Time

A Date/Time value is represented as an integer number of milliseconds since Unix epoch (1970-01-01 00:00:00 GMT). Date functions will help you convert those numbers to readable Text values.

A Date value (without the time component) is represented in the same way as the date/time value, calculated for the midnight of the specified day – *in the user's current time zone*. ⚠ This means that expressions involving Date values (for example, using Jira's Due Date field) may produce different results for different users if they are in different time zones.

## Duration

A Duration value is represented as an integer number of milliseconds. Duration functions help transform duration values into readable Text values.

## Boolean

A boolean value, such as returned from comparison functions, is represented by a number 1 if true, or 0 if false.

Additionally, all values can be converted to a boolean value based on whether they are "truthy". (See [Value Conversions](#).)

## Item

An item value represents an object that you can potentially have as a row in a structure – a Jira issue, project, sprint, version, user, status, priority and others. It can also represent purely Structure-owned objects like folders.

Items have properties that you can access by writing `item.propertyName`, or `(item-resulting expression).propertyName`. The types of items, their properties and their type is listed in [Item Property Reference](#).

## Special conversion of Item values

Item values always can be used as Text. Each item will then be represented by a text – an issue by issue key, a project by its name, etc.

Also, items may be compared using `UMAX` and `UMIN` functions, which will respect the "natural" order for the items. Most items will be compared alphabetically, but some will have a predefined order, like `Priority` items.

## Special treatment of Item values by generators

Similar to max/min operations, if a formula column produces Item values, then you can sort by that column (or use `Attribute Sorter`) to reorder the structure according to the natural order of the resulting items.

If an `Attribute Grouper` is configured with a formula that produces Item values, the groups will become those items.

## Array

An array value is a list of other values. Each value in an array is called an element of the array.

Each value in an array may be of any type – you can have an array of texts, array of items, array of arrays, and so on. An array can contain different types of values in each element – the first one could be a number, the second one an item, and the third one an array of some other elements.

The values in an array may be accessed by their index. The first value has index 0, the second value has index 1, and so on.

There's a number of array functions that allow you to write formulas that deal with multiple values – see [Array Functions](#).

## Creating an array

You can create an array by calling the `ARRAY` function: `ARRAY(element1, element2, element3, ...)`

When you need to change an array, you apply some transformations to an existing array – most frequent being `FILTER` and `MAP`. This effectively creates a new array.

## Accessing array elements

You can access an element of an array by its index, using the `GET` function: `array.GET(index)`

However, it's typically not needed, as the calculations with arrays are done mostly with `FILTER`, `MAP` and `REDUCE`.

## Special treatment of Array values by the Attribute Grouper generator

When an array value is returned by a formula to the `Attribute Grouper` generator, it will create multiple groups – one for each non-empty, distinct value returned by the formula.

## User Function

A user function value (sometimes called a "lambda value") represents a piece of Expr code that defines a function – see `User Functions` below.

Typically, you create a user function value to apply it to an array via one of the system functions, such as `FILTER` or `MAP`.

For example, consider the following code:

```
worklogs.FILTER($.author = ME()).MAP(w -> w.timeSpent).REDUCE((a, b) -> a + b)
```

There are three user functions here, `$.author = ME()`, `w -> w.timeSpent`, and `(a, b) -> a + b`. The first defines the condition for filtering work logs, the second defines how to transform each work log into a number, and the third one defines how to aggregate multiple numbers into one.

Each of these three pieces of Expr are evaluated and are represented by a value of the "User Function" type, and passed to `FILTER`, `MAP`, and similar functions.

As with any other value, a User Function value can be assigned to a variable. A more traditional function definition form could be used for that. The following two definitions are identical:

```
WITH square = x -> x * x :  
...  
WITH square(x) = x * x :  
...
```

## Key-Value Map

A key-value map is produced by some of the Expr functions, and is a collection of pairs, where "key" is the property name and "value" is the property value.

Currently, only `GROUP` function creates a key-value map, with key `group` mapping to the value the array is grouped by, and key `elements` mapping to an array of the matching elements.

To access the value, you need to write `kvm.keyName`, where `kvm` is a local variable or an expression that represents the key-value map.



Keys are supposed to be well-known; there's no way to iterate over all keys.

There's no way for the user to create an arbitrary Key-Value Map; this type is intended only for some system functions.

## Errors

Errors are special values, which indicate that the calculation of some part or the whole of the expression encountered a problem.

The list of possible errors is available at [Expr Error Codes](#).

Normally if an error occurs somewhere in the formula, the result of the whole calculation is an error as well. But, if receiving an error from a part of the expression is legitimate, you can use the **IFERR** and **ISERR** functions to handle it.

## Value Conversions

Each Expr function expects a value of a certain type to be passed as each of its parameters. (See [Expr Function Reference](#).) When a value of a different type is encountered, an automatic conversion is attempted. If conversion is not possible, an error will result.

The conversion rules follow some basic principles:

- Best effort is made to convert simple types, such as converting a text to a number
- An item can be converted to text that represents it
- Any value can be converted to an array of one element (array "wrapped around" a value)
- An array with just one value can be converted to that value ("unwrapping" the array)

There are certain more specialized cases related to Text/Joined parameters, parameters marked as /Each and parameters of the User Function type.

The table below summarizes all the conversion rules.

Actual type Required type	Number	Text	Item	Array	Key-Value Map	undefined
Number	pass as is	<ul style="list-style-type: none"><li>empty string undefined</li><li>try to convert to a number</li><li>if unsuccessful: error</li></ul>	use the item's text representation, according to the item type, and then use the rules for converting Text to Number	<ul style="list-style-type: none"><li>empty array undefined</li><li>has only one element try to convert that element</li><li>otherwise: error</li></ul>	error	undefined
Integer	<ul style="list-style-type: none"><li>use if an integer value</li><li>otherwise: error</li></ul>	same as above, and try to convert to integer				
Date	<ul style="list-style-type: none"><li>same as above: timestamps are represented as "unix epoch milliseconds"</li><li>dates are represented as a timestamp of the corresponding day's midnight in the user's time zone</li><li>using non-integer values may result in error</li></ul>					
Boolean	<ul style="list-style-type: none"><li>false if zero</li><li>true otherwise</li></ul>	<ul style="list-style-type: none"><li>false if empty or only whitespaces</li><li>true otherwise (including "0!")</li></ul>	true	<ul style="list-style-type: none"><li>false if empty</li><li>true otherwise (including an array with 0 as an element)</li></ul>	true	false

<b>Text</b>	convert to text	pass as is	use the item's text representation, according to the item type	<ul style="list-style-type: none"> <li>empty array undefined / empty text</li> <li>has only one element try to convert that element</li> <li>otherwise: error</li> </ul>	error	undefined / empty text
<b>Text /Joined</b>	<i>same as above</i>			<ul style="list-style-type: none"> <li>collect a <b>Text</b> (not a Text /Joined!) value for each non-undefined element</li> <li>join elements into one string with " , " as a separator</li> </ul>	<i>same as above</i>	
<b>Array</b>	array with one element	array with one element	array with one element	pass as is	array with one element	undefined / empty array
<b>Item</b>	error	error	pass as is	<ul style="list-style-type: none"> <li>empty array undefined</li> <li>has only one element try to convert that element</li> <li>otherwise: error</li> </ul>	error	undefined
<b>Key-Value Map</b>	error	error	error	<ul style="list-style-type: none"> <li>empty array undefined</li> <li>has only one element try to convert that element</li> <li>otherwise: error</li> </ul>	pass as is	undefined
<b>Any</b>	any value works					

## Text to Number Conversion

Some functions expect their arguments to be number values. In case an argument is a text value, we try to interpret it as a number. This can be useful if the value comes from a variable that represents a text custom field, which contains numbers — e.g., imported from some external system.

If conversion is successful, that number is used as the value for that argument. If conversion is not successful, functions can either produce an error, ignore that argument, or substitute some default — it depends on the function; see [Expr Function Reference](#) for details.

The first step is to accommodate for variations in number formatting. Conversion supports these formatting symbols:

- Decimal fraction separators: comma (,), dot (.)
- Digit group separators: comma (,), dot (.), apostrophe ('), space ()

Conversion expects that the text contains 0 or 1 decimal mark, and 0 or more group separators of the same kind. If the text contains any other formatting symbols, conversion fails. Decimal mark must come after all group separators, otherwise conversion fails.

If the text contains only one formatting symbol, and it's a dot (.), it is always treated as a decimal mark. If the text contains only one formatting symbol, and it's a comma (,), then it is treated as a decimal mark if a comma is used as a decimal separator mark in the [Jira default language](#); otherwise, it is treated as a group separator. For instance, if the default Jira language is English, "101,112" will become 101112, whereas if it is German locale, it will be 101.112. And regardless of language, "1 100,23" will become 1100.23: space is interpreted as a group separator, and comma can only be the decimal fraction separator here.

If the group separator is a dot (.), then all groups except the first one must have 3 digits; otherwise, conversion fails.

After determining decimal mark and group separator symbols, conversion removes all group separator symbols and replaces the decimal mark with a dot. Note that if text contains several whole numbers separated by spaces, conversion will think it is one number, for example, "10 11 12" will become 101112. Similarly, "10,11,12" will become 101112.

The final step of conversion is to recognize the resulting text as either Expr's literal number representation or scientific or engineering notation. Examples:

```
0.239
-1.32e5
12e-3
```

## Conversion to Boolean: Falsy and Truthy Values

A value is *falsey* if it is:

- undefined,
- number 0,
- an empty text value (" " or ' '), or a text value that contains only space characters,
- an empty array.

All other values are *truthy*.

When converting to a Boolean, truthy values become true and falsy values become false.

By convention, when functions or logical operators need to construct a truthy value, they use the number 1.

## Text vs. Text/Joined

When a function declares that it requires "Text/Joined" value, it means that the value will be converted to a text, with an additional special handling of the array type:

- If it's an empty array, or an array with only one element, the conversion will be the same as for "Text" type.
- If an array with multiple values is passed, then a) each element of the array will be converted to a Text value, b) all these texts will be joined together with a comma as a separator.

Here's an example illustrating the difference when fixVersion is passed as a parameter - notice that in the third row, there are multiple values in fixVersion (because it's an array), so Text and Text/Joined are treated differently:

fixVersion	Function accepting Text will receive	Function accepting Text/Joined will receive
(no value)	undefined	undefined
v1	"v1"	"v1"
v1, v2	error	"v1, v2"

## Passing Implicit User Function as a Parameter

You can always pass an implicit User Function (the one containing the "\$" symbol) as an argument to a function. This will result in the call to this function to become an implicit User Function value itself.

For example, consider the following expression:

```
fixVersions.FILTER(YEAR($.releaseDate) = 2021)
```

Function **YEAR()** expects a date, but it receives a User Function instead (**\$.releaseDate**), which produces the release date for each passed version. As a result of applying **YEAR()** to that User Function, we will get another user function, which produces the *year of the release date* for each passed version.

This logic applies only to the implicit User Functions, defined with the \$ sign.

The functions that expect a User Function parameter, like **FILTER**, are exclusions from this rule.

## Variables

Variables (also known as free, or externally set variables) represent some values that will be fed into the formula for each Structure row that the formula will be applied to.

For example:

```
IF priority = "Blocker" : parent.estimate + x
```

In this formula, "priority", "parent" and "x" are all variables – they vary from one row to another. (They will not change the value while calculating the expression for a single row.)

There's no need to declare a variable, you can immediately start using it. All valid identifiers that are used like variables will be treated as such.

Each formula is expected to contain at least one variable – otherwise, the result will be the same for each row.

## Variable to Attribute Mapping

Each variable should be mapped to a valid attribute – such as a Jira field or a Structure attribute, so when an expression is calculated for a particular item, the value of that attribute becomes the variable's value.

If you use one of the well-defined variable names, it will be automatically mapped to the corresponding attribute. See [Standard Variable Reference](#).

If you use an arbitrary variable name, such as "x", you will need to map it as described on the [Mapping Variables](#) page.

If a variable is not mapped, or if the item does not support the mapped attribute, the value of the variable will be the `undefined` value.

## "this" Variable



One of the well-defined variable names is "this". It is mapped to an attribute that provides a value of Item type, representing the item for which the formula is being calculated.

This may come in handy in certain cases. For example, to analyze issue links and pick the "other side" of a link, regardless of whether it's an incoming or an outgoing link:

```
• issueLinks.MAP(IF $.source = this : $.destination ELSE $.source)
```

Alternatively, you can use "item" with the same meaning.

## Local Variables

Local variables are similar to Variables, but they are not mapped to an item's attribute or Jira field, but rather defined and calculated right in the expression.

The declaration syntax is the following:



Note the colon (":") that separates the expression assigned to the variable and the expression where the variable is used.

A few facts about local variables:

- *ExpressionWithLocalVariable* may start with another local variable definition, so you can introduce many local variables in sequence. When defining a second variable, you can use the first variable already defined, and so on.
- Local variables can "shadow" previously defined local and free (mapped) variables with the same name. If you write **WITH priority = 10: <expression>**, then when calculating <expression>, the value of **priority** will be 10, even if there was a variable attached to the issue's priority in the enclosing scope.
- The **WITH...** construct is itself an expression, so you can use it, enclosed in parentheses, anywhere an expression can be used. The name defined in this expression is not visible outside the **WITH...** expression.

## Immutability

Expr language constructs are immutable. Once a local variable is defined, it cannot change its value. (So, in fact, calling it a "variable" is not exactly correct. Although, if a local variable depends on external variables, which vary from item to item, the local variable itself will also vary from item to item.)

So if you're building your formula and you need to take a number of values through a series of calculations, you may need to use multiple local variables, going through each step and assigning each intermediate result to another local variable.

## Operators

Operators allow writing formulas in a convenient way using traditional logical and arithmetic operations. Each operator has one (unary operators) or two operands. Each operand could be a variable, a number, or just any expression, which sometimes will need to be in parentheses.

The operators translate into calling the corresponding functions on operands as arguments. If an operand expression evaluates to an error, then the operator's result will also be an error.

Available operators are:

Operator(s)	Symbol(s)	Priority	Type of Operands	Result Type	Corresponding Function(s)
Logical negation	NOT	7	Any	Number (Boolean)	NOT
Unary sign	+ -	7	Convert to Number	Number	SUM, MINUS
Multiplication and division	* /	6	Convert to Number	Number	MUL, DIV
Addition and subtraction	+ -	5	Convert to Number	Number	SUM, MINUS
Concatenation	CONCAT	4	Convert to Text/Joined	Text	CONCAT
Equality/Inequality check	= !=	3	Any	Number (Boolean)	EQ, NE
Numeric Comparison	< > <= >=	3	Number	Number (Boolean)	LT, GT, LE, GE
Logical AND	AND	2	Any	Any(*)	AND
Logical OR	OR	1 (lowest)	Any	Any(*)	OR

The operators are listed in their priority order. The priority is important when an expression could allow different interpretations about which operators are applied first.

For example, in an expression `progress + parent.progress * weight < threshold AND priority != "Blocker"`, the multiplication is executed first, then the addition, then the comparisons, and then the logical AND. If you'd like to alter the order of operator application, use the parentheses.

## Logical Negation (NOT)

To negate a logical value or expression, use the **NOT** operator. Instead of **NOT**, an exclamation mark (!) can also be used.

The operator produces 0 if the operand is a truthy value, and 1 otherwise. Therefore, it may be applied to any value.

If the value negated is an expression with other operators, it should be contained in parentheses.

Examples:

- **NOT resolved**
- **NOT (storyPoints > 0 AND storyPoints < parent.maxStoryPoints)**

## Unary Sign (+ -)

The operator first attempts to convert the value of an expression to a number. If conversion succeeds, + produces this number, and - produces the negated number.

If the conversion to a number fails, and the value of the expression is falsy, the negation produces `undefined`. Otherwise, it produces an error.

## Arithmetic operators (\* / + -)

Arithmetic operators are: addition (+), subtraction (-), multiplication (\*) and division (/).

Multiplication and division have precedence over addition and subtraction.

These operators convert their arguments to numbers. A non-empty, non-number argument would produce an error. Falsy non-number values are treated as zero.

Examples:

- `"" + 1 1`
- `"foo" + 1 error`
- `"" * 1 0`
- `"foo" * 1 error`
- `"" - 1 -1`
- `1/0 error`

If any subexpression produces an error, the operator produces the same error.

## Concatenation (CONCAT)

The concatenation operator converts each operand to a text value (Text/Joined, to be precise) and creates a new text by joining them together. It is identical to calling the `CONCAT` function on its operands:

```
value1 CONCAT value2 CONCAT value3 = CONCAT(value1, value2, value3)
```

The operator can be used to increase the readability of a formula.

Note that `CONCAT` is applied after all the arithmetic operators but before comparisons and logical operators.

## Equality check (= !=)

The "equals" operator (=) checks that both arguments are "essentially the same value". The "not equals" operator (!= or <>) produces the inverse value.

The comparison rules are based on the type of the values compared.

Equality operator will return true (1) if any of the following conditions hold:

- Both values are **Undefined**.
- One value is a **Number**, and the second value is the same number, or can be converted to the same number.
- One value is a **Text**, and the second value can be converted to Text (using Text/Joined), and both values are "essentially the same".
  - The differences in letter forms and leading and trailing whitespace are ignored (thus `" cote "` = `"côte"`).
- One value is an **Item**, and the other value is the same item. (Note that if the other value is a Text, both values can be compared as text values.)
- One value is a **Key-Value Map**, and the other value is also a key-value map with the same contents (these equality rules applied to all elements).
- One value is a **User Function**, and the other value is exactly the same user function (not just having the same logic).
- One value is an **Array**, and either of the following is true:
  - The other value is also an **array**, has the same number of elements and the elements are respectively equal.
  - The other value is **undefined** and the array does not contain any non-undefined elements.

- This array contains only **one element** and this element is equal to the other non-array value we are comparing the array with.

In all other cases, the equality returns false (0).



Note that you can compare an item to a text, for example,

```
IF project = "My Project" : ...
```

The item will get converted to a text using value conversion rules described earlier.



If one value is a number and the other value can be converted to a number, both values are treated as numbers. However, if both values are text, they will be treated as text, even if both can be converted to a number. You can use the [NUMBER](#) function to force a value to be numeric.

- `3.4 = 3.40` 1
- `3.4 = "3.40"` 1
- `"3.4" = "3.40"` 0
- `NUMBER("3.4") = "3.40"` 1

## Numeric Comparison (< > <= >=)

The ordering / comparison operators work on numbers only:

- < (less than)
- > (greater than)
- <= (less than or equal)
- >= (greater than or equal)

If either of the values is text, the operator attempts to convert it to a number. If the conversion fails, the result is an error.

If any value is undefined, strict operators (<, >) produce 0. Non-strict (<=, >=) produce 0, unless *both* values are undefined (because they are equal).

## Logical operators (AND OR)

The logical operators are used to combine other logical conditions, or to pick an alternative or a conditional value.

- OR also can be written as " | " or " | "
- AND also can be written as "&&" or "&"

When both operands are Number(Boolean), then the operators perform the corresponding boolean operation.

However, you can use these operators with non-boolean operands in a "short circuit" way, based on whether the operands are "truthy" (see above).

- `a OR b` – if "a" is truthy, "b" is not evaluated and the result of the operation is "a"; otherwise the result of the operation is "b".
- `a AND b` – if "a" is falsy, "b" is not evaluated and the result of the operation is "a"; otherwise the result of the operation is "b".

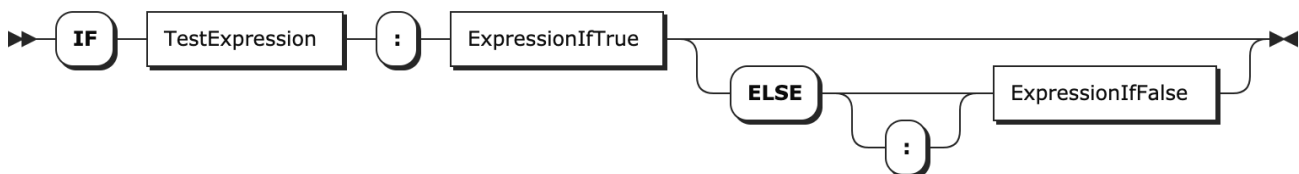
Note that "b" is not evaluated in case the result of the operation is equal to "a". This might be important in case calculating "b" could result in an error.

Examples:

- `assignee OR "UNASSIGNED"` – This will produce either the issue's assignee user key or (if the issue is unassigned) the text value "UNDEFINED".
- `!assignee AND status = "OPEN"` – This will produce 1 if the issue is unassigned and in status OPEN, and 0 otherwise.
- `count AND total / count` – This will produce some average number (total / count), unless count is 0 – in this case the result will be 0. Note that there will be no division by zero error.

## Conditional Expression

A conditional "IF" expression allows you to switch between two expressions, based on whether a condition is true (truthy) or false. It is identical to calling the IF function with two or three arguments.



The "ELSE" part, as well as the colon (":") after ELSE are optional. If the ELSE part is omitted, and the test expression evaluates to false, the result is undefined.

If you use nested IF expressions with only one ELSE, the ELSE part applies to the innermost IF. We recommend using parentheses to make it clear which IF it applies to.



You can use indentation to make the formula with nested IFs more readable – but the indentation has no effect on how the formula is parsed. Use parentheses!

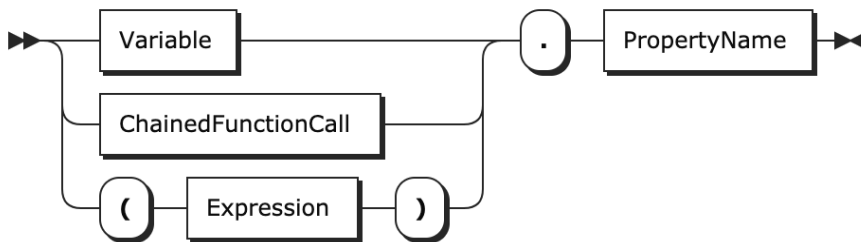
Examples:

```
IF assignee = ME() : "mine!"
---
IF dueDate < NOW() : "overdue!" ELSE: ""${DAYS_BETWEEN(NOW(), dueDate) - 1} days left!""
---
IF priority = "Critical":
IF dueDate < DATE_ADD(TODAY(), 7, "days"):
    "critical and urgent"
ELSE:
    "critical and not urgent"
```

## Property Access

Some Expr values – Items and Key-Value Maps – contain "properties", named values that are parts of the bigger value. For example, a Version item contains properties such as "name", "description", "releaseDate". And function GROUP returns a value that is an array of Key-Value Maps, each map having "group" and "elements" properties.

To access a property, use the dot ("."), followed by the property name. The names are case-insensitive.



Examples:

- `fixVersion.releaseDate`
- `project.lead`
- `epicStories.storyPoints`

An alternative way to access a property is by using the `ACCESS()` function – in this way, the property name itself may be calculated:

- `parentTask.ACCESS("Story Points")`

⚠ If the value does not have the requested property, or if the value is neither an Item nor a Key-Value Map, the resulting value is `undefined` – not an error! (See also a note about arrays below.) ⚠



There's no way to check if a value contains a certain property (other than try and access it), or list all available properties for an item.

## Accessing Custom Fields via Properties

In most cases a formula will refer to the issue's custom fields directly by name – for example, the formula `impact / storyPoints` uses the values of two custom fields, Impact and Story Points, of the currently calculated issue to calculate the benefit-to-cost ratio.

You can, however, use a formula to access other, related issues, and their corresponding custom fields – for example, the formula `impact / (storyPoints + subtasks.storyPoints.SUM())` calculates a similar value but takes into account the cost (in story points) of all the subtasks.

Note that the actual custom field is named "Story Points", with a whitespace. When you access a property of an issue item, Structure tries to match the property name with available custom field names using loose rules, dropping whitespace and any non-identifier-friendly characters. The property name is also case insensitive.

You can also use the `ACCESS()` function to specify the name of the field precisely, or use "customfield\_NNNNN" property name to identify a field by its ID.

- `parent.storyPoints`
- `parent.ACCESS("Story Points")`
- `parent.customfield_10000`



You can use `this` variable to access properties of the currently calculated item. Normally it's not needed, since the same values are available as corresponding variables.

## Accessing Property of Each Element in an Array

You can apply property access to an array of values. Expr will then apply property access to each element and return the result as an array.

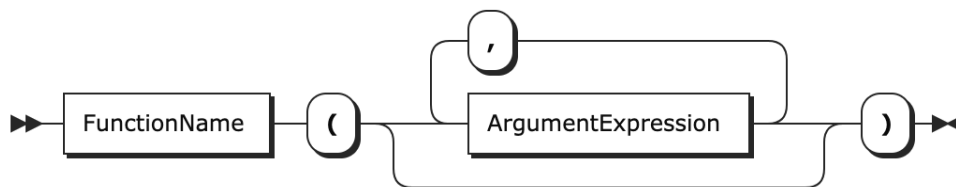
For example:

- `fixVersion.releaseDate` – will return an array of release dates
- `worklogs.author.UNIQUE()` – will return a list of people who logged work
- `subtasks.remainingEstimate.SUM()` – will return a total remaining estimate from the subtasks

In the resulting array, all `undefined` results will be removed and the array will be flattened. This is the same behavior as shown by functions with `/Each` parameter type.

## Functions

A function takes zero or more values, and produce another value. A function call consists of a function name (an identifier), followed by its arguments enclosed in parentheses. An argument can be any expression. Different arguments are separated by commas (,) or semicolons (;) — for one function call, all separators must be the same.



Examples:

- `NOW()`
- `ROUND(storyPoints / 10)`
- `FILTER(sprint, $.state = "active")`
- `MAKE_DATETIME(2017; 12; 31; 23; 59; 59)`
- `myUserFunction("argument1", "argument2")`

A function call can evaluate only some or even none of the arguments, depending on the function. This is useful for functions that perform choices. For example, in an `IF` function, the argument that wasn't chosen is not evaluated, so the whole expression doesn't produce an error when that argument produces an error.

## System and User Functions

System functions are provided by Structure. The functions are listed in the [Expr Function Reference](#). Each function expects a certain number and type of parameters.

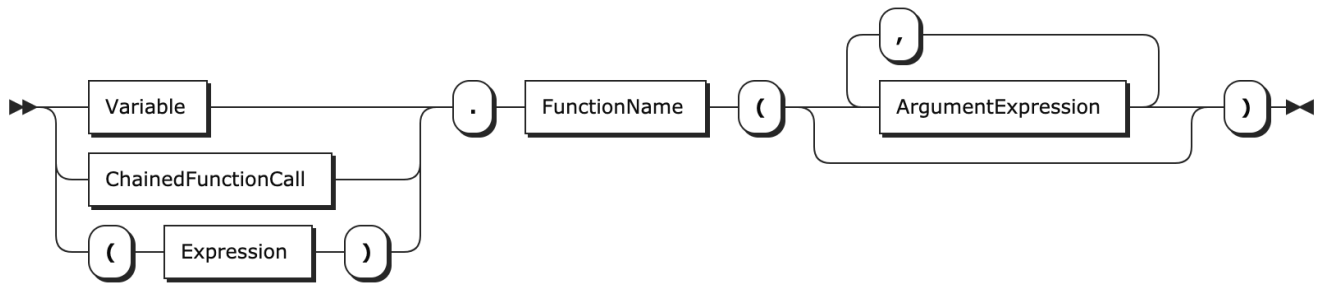
User functions are explained in the next section. A user function is called using the same syntax as a system function.



User functions can take any number of arguments, regardless of how many arguments are declared. If a parameter was declared, but a value was not provided when calling the function, the parameter's value will be `undefined`.

## Chained Function Calls

A different way to call a function is by "chaining" it to its first argument by adding a period ("."), a function name, parentheses, and any additional parameters, if any.



For example, `sprint.FILTER($.state = "active")` is the same as `FILTER(sprint, $.state = "active")`.

This allows nice, readable expressions, where a value is sequentially transformed by applying functions to the result of a previous function call:

- `affectsVersion.MAP($.releaseDate - $.startDate).MAX()`
- `linkedIssues.FILTER(x -> NOT x.resolution).MAP(x -> x.remainingEstimate).SUM()`



Unlike some other languages, in Expr any function may be written in the chained syntax, regardless of what the value is.

## Applying Functions to Arrays

When a function is applied to an array value (meaning that an array is passed as the first argument of the function), the result may be calculated in a number of ways, depending on which function is called and what type of argument it expects.



You can always apply a function to each element of an array using the MAP function. For example:

- `array.MAP(SOME_FUNCTION($))`

The cases below relate to cases where an array is passed directly as an argument: `SOME_FUNCTION(array)`.

## Using an Array Function

There are special functions that expect an array as their first argument – `FILTER`, `MAP`, and [others](#).

There's no special behavior in this case – an array is expected. In fact, if the value passed is not an array, it will be converted to either an array of one element (containing that value), or an empty array if the value is undefined.

## Passing Array as an Argument of Text/Joined Type

If a function declares that it expects "Text/Joined" as an argument, then the system will try to convert an array into a text value. See the "Text vs Text /Joined" section above.

For example:

- `CONCAT("Versions: ", fixVersion) "Versions: v1, v2, v3"`

## Passing Array as an Argument of /Each Type

If a function declares that it expects "Number/Each" or "Text/Each" or any other "/Each" type as an argument, then it would work on that simple type, but if an array is passed, it will apply its logic to each element in that array. The result of calling this function will be an array, where each element is a result of applying the function to the original element.

For example:

- `UPPER(fixVersion) ARRAY("V1", "V2", "V3")`



In addition, when applying a function to an array in this way, the resulting array is "flattened" (elements from any sub-arrays moved to be the elements of the top array) and "compacted" (all undefined elements are removed).

## Passing Array to a User Function

You can call a user function and pass an array as an argument. No special handling takes place – just as with a system function expecting an array.

## All Other Cases

If a system function does not expect an array, but it is passed as an argument, it will try to convert it to the value type it expects. A one-element array will be converted to its single element and an empty array will be converted to `undefined`. See "Value Conversions" above.

If the conversion is not possible, the result will be an error.

## User Functions

User Functions are functional *expressions*, defined by the user. (They could also be called "lambdas".) User functions are helpful when the user needs to apply some repetitive action, or to pass an action to be applied to each item in an array.

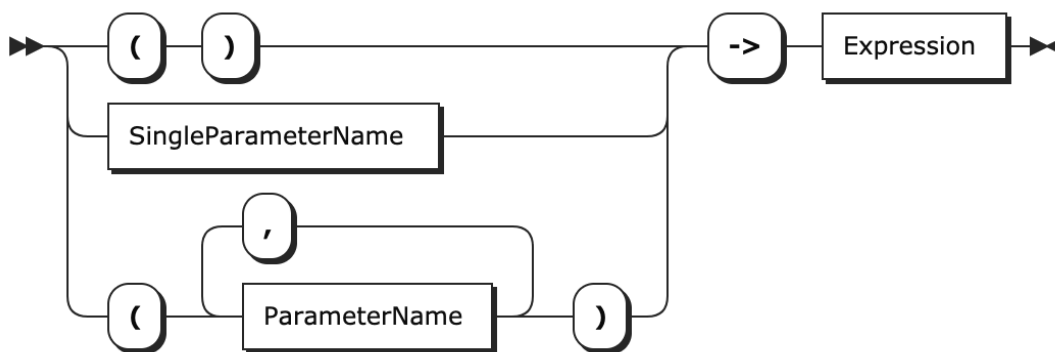
A User Function contains a list of parameters and then an expression that is calculated for these parameters.

A User Function is a *type of value*, so you can assign a user function to a local variable, or pass it to some higher-order function as a parameter.

There are three ways to define a user function.

## Functional Expression

A functional expression is the canonical form for user functions. It contains a list of parameters in parentheses, followed by the "maps to" symbol ( $\rightarrow$ ), followed by the expression calculated by the function. When there's only one parameter, the parentheses can be omitted.



Examples:

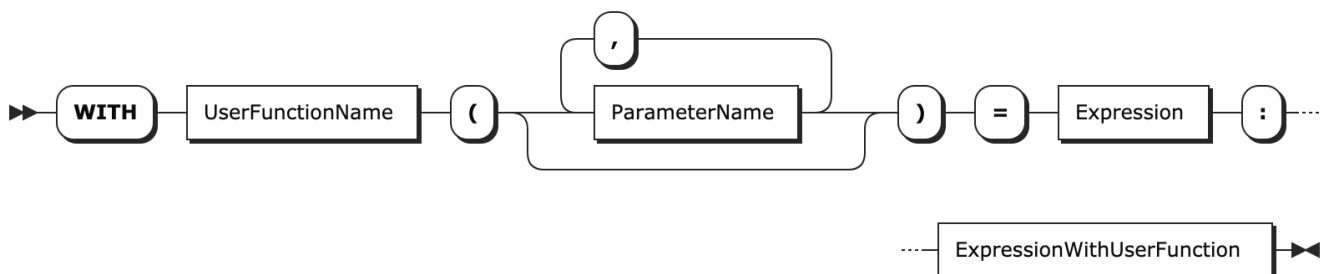
- `() -> START_OF_MONTH(NOW())`
- `(x) -> x * x`
- `version -> version.releaseDate - version.startDate`
- `(s1, s2) -> s1 CONCAT " " CONCAT s2`

All these examples evaluate to a "User Function" value type, which can be assigned to a variable:

- `WITH square = x -> x * x : ...`

## Traditional Function Definition

A more traditional function definition looks similar to a variable definition, only the variable is followed by a list of parameters in parentheses, and the expression is based on those parameters.



To rewrite the examples above:

- `WITH currentMonth() = START_OF_MONTH(NOW()) : ...`
- `WITH square(x) = x * x : ...`
- `WITH versionDuration(version) = version.releaseDate - version.startDate : ...`
- `WITH joined(s1, s2) = s1 CONCAT " " CONCAT s2 : ...`

These declarations are identical to the corresponding examples in the previous section, with local variables assigned to the corresponding User Function values.

## Implicit Functional Expression (\$)

Most of the time when we're creating a formula with an array, we need to apply some kind of operation to each element of the array. Implicit functional expressions help define the corresponding user function easily by having "\$" denote "each element".

For example:

- `versions.FILTER($.startDate < NOW())`
- `issueLinks.FILTER($.type = "Relates").MAP($.destination)`
- `worklogs.UMAX_BY(IF $.author = ME() : $.timeSpent)`

In each case, the expression with "\$" is transformed into a User Function with a single parameter, which is then substituted for \$. So, the last example from the list above is identical to:

- `worklogs.UMAX_BY(w -> IF w.author = ME() : w.timeSpent)`



When reading these expressions, you can say "each" when the dollar sign is encountered.



An implicit user function must always be used in an argument to a system function, which expects a user function. Otherwise, it won't be accepted.

For example – here's how we can filter an array to contain only even numbers:

Correct	Incorrect – Parse Error
<code>ARRAY(1, 2, 3).FILTER(MOD(\$, 2) = 0)</code> <i>... or, alternatively ...</i> <code>WITH even(e) = MOD(e, 2) = 0 : ARRAY(1, 2, 3).FILTER(even)</code>	<code>WITH even = MOD(\$, 2) = 0 : ARRAY(1, 2, 3).FILTER(even)</code>

## Calling User Functions

If a User Function is assigned to a variable, you can call it in your expression in the same way you call a system function.

- `WITH square(x) = x * x :  
square(impact) / square(cost)`

You can also use the chained function call notation:

- `WITH square(x) = x * x :  
WITH fquare(x) = x.square().square() :  
storyPoints.fquare()`



Note that you cannot invoke a functional expression unless it is assigned to a local variable. The following will produce an error: `(x -> x * x) (3)`

## Function Name Collisions

Both system functions and user functions are invoked in the same way – `FUNCTION_NAME(arg1, arg2, arg3, ...)`, or with a chained call syntax – `arg1.FUNCTION_NAME(arg2, arg3, ...)`. This leaves a potential for the user to define a function that has the same name as a system function.

⚠ When Expr encounters a function call, first it looks up if there is a *system* function of that name. ⚠

The system function will be called even if there's a local variable of the same name. To protect the user from name collisions, Expr will show an error if you try to define a function with a name that matches a system function name. (However, it won't be able to detect the collision if a local variable is defined through a series of assignments of a functional expression.)

You *can* define a local variable of any other type with a name identical to a system function's name.

Works as expected	Error
-------------------	-------



```
// Using "SUM" as a local variable,
// but "SUM()" is also a system
function
// and "SUM{}" is an aggregate
function.
WITH SUM = cost + parent.cost :
SUM(SUM, SUM { cost })
```

```
// Cannot define a user function with a name collision.
// Note that the language is case-insensitive: "SUM" and "sum" are
the same.
WITH sum(issue) =
    issue.timeSpent + issue.parent.timeSpent :
...
```



Note that the function name collision resolution provides potential challenges when upgrading to a newer version of Structure, if that version introduces new system functions.

Let's say you have defined a user function `LAST_COMMENT()` in your formula and used it successfully in an older Structure version. If the newer version of Structure adds a system function `LAST_COMMENT()`, that formula will likely stop working after the upgrade, and you will need to rename the user function.

To minimize the probability of this happening, we suggest naming your user function in a way that makes potential collision unlikely. It could be a name that is very specific to your configuration, or you can always prepend the name with an underscore – in our example, call it `_LAST_COMMENT()`.

## Aggregate Functions

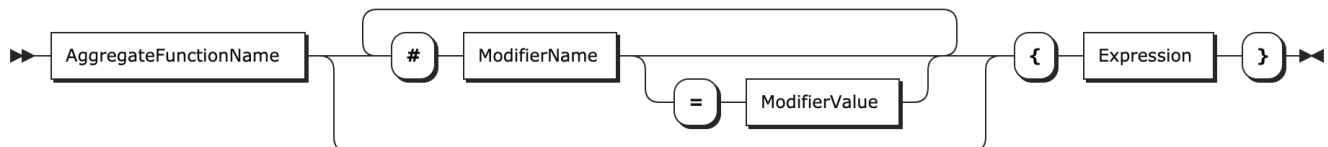
An aggregate function calculates values for some other rows in the structure (for example, for all sub-items), aggregates these values (for example, adds them together), and produces the resulting value to be used in the formula.

For example:

- `SUM#children { storyPoints }` – calculates total story points for all the child issues.
- `PARENT { fixVersion }` – provides the value of the `fixVersion` field from the parent's issue.
- `VALUES { components }` – collects all distinct components that are set for any of the sub-issues of the current row.

Aggregate functions allow you to calculate complex values that depend not only on the "current" item, but also on other items in relation to it.

An aggregate function starts with a name, optionally followed by modifiers, then curly braces ("{}"), and inside them – an "inner expression", which will be calculated for some other rows. You can use whitespace between any elements of the aggregate function calls.



The inner expression may return any type of value – number, text, array, and others – *except a user function*. You cannot pass a user function from the inner formula into the outer formula.

Available Aggregate Functions are listed in [Aggregate Function Reference](#).

## Aggregate Function Modifiers

An aggregate function may have one or more modifiers that govern the aspects of the function's execution. Each modifier starts with hash sign ("#"); then comes the modifier's name; optionally followed by the equals sign ("=") and a value, which can be a string or a numeric constant. If a value is omitted, it is assumed to be 1 (a representation of *true* in Expr).

Examples:

- `SUM#all { cost }`
- `SUM#all#leaves { IF type = "story": storyPoints }`
- `JOIN#separator=", " { key }`
- `JOIN #separator=", " #fromDepth=0 #toDepth=-1 { key }`

Each aggregate function supports a specific set of modifiers, not all of them. Using an incompatible modifier will result in an error. To learn more about available modifiers and their restrictions, see [Aggregation Modifiers](#).

## Sharing Values Between Outer and Inner Formulas

It's important to understand that the formula inside an aggregate function – the inner formula – is calculated fully separately from the "outer" formula. Both formulas will share variable mappings (to attributes), but any local variables and user functions defined on one side will not be accessible from the other side.

Will not work!	Correct version
<pre>// Cannot use "total" on line 4!  WITH total = SUM#children { storyPoints } : WITH median = MEDIAN#children { storyPoints / total } : ...</pre>	<pre>WITH median = MEDIAN#children {   WITH total = PARENT { SUM#children { storyPoints } } :   storyPoints / total } : ...</pre>

As you can see from the example above, you may need to use nested aggregate functions instead.

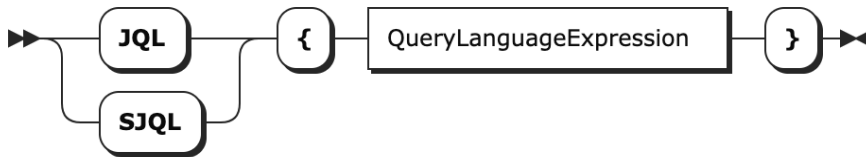
## Using Formulas with Aggregate Functions in Generators and Transformations

Note that when you use an aggregate function, it relies on the existing structure to figure out the parent item, child items, and other related items addressed by the formula. When the formula is used to build or transform a structure, the hierarchy and items that the aggregate functions "see" will correspond to the structure that exists immediately before the generator or the transformation is applied.

Therefore, when using a formula with Attribute Filter, Attribute Grouper, Attribute Sorter and other generators, you should apply aggregate functions carefully, understanding what is the preceding structure before the generator is applied. For example, if you use a grouper on a folder with a flat list of issues, the formula in the grouper will see issues and the folder will be their parent item – there are no groups yet!

## Embedded Queries

You can embed JQL and [S-JQL](#) in a formula to check if the item (the one the formula is being calculated for) satisfies the condition of the query – that is, it will be a part of the query result.



The syntax is similar to calling an aggregate function:

- IF JQL { assignee in membersOf("Team-Alpha") } : ...
- IF NOT SJQL { descendant of folder("Excluded") } : ...

The result of evaluating JQL { } or SJQL { } is always 0 (false) or 1 (true).

When SJQL is used, it is always applied to the current structure. (Or the one being generated – see the note below.)

Note that, unlike aggregate functions, these constructs do not use Expr, but rather another other languages, as the inner expression. Use the corresponding documentation as a reference for JQL and S-JQL.



The embedded queries are calculated separately from the Expr formula they are used in. Therefore, you cannot use any values from the Expr formula inside a JQL or SJQL query or vice versa. Also, you cannot check the query match for any other item except the one the formula is being calculated for.

In other words, the only data that is passed between an embedded query and the outer Expr formula is 1 or 0 depending on whether the current item matches the query.

## Using S-JQL in Generators and Transformations

An S-JQL query usually depends on the structure it's being calculated for. So, similar to Aggregate Functions, when you use a formula with an embedded S-JQL in a generator or a transformation, the query will be applied to the underlying "preceding" structure, that exists before the generator is applied.

## S-JQL Query Performance

Structure optimizes the calls to embedded queries. A query will be run only once for multiple items that the formula is being calculated for, and the result will be checked separately in the calculations for each row.

That said, the JQL itself may potentially be an intensive calculation, if it uses JQL functions or historical conditions like WAS. Please be careful when trying this JQL in a formula, and watch for how long the value is being calculated before publishing the formula for other users. Normally, calculating a "heavy" formula should not prevent users from doing other things in Structure (including using other columns with formulas), but it can place some stress on the Jira server.



Please avoid using the `structure()` JQL function in JQL or S-JQL that is being embedded in a formula.